Programmare con le interruzioni

G. Lettieri

10 Aprile 2018

Proviamo a modificare il programma di esempio della tastiera (Vol. II) in modo da utilizzare il meccanismo delle interruzioni. Vogliamo mostrare quali sono i problemi a cui bisogna prestare attenzione quando si scrivono questo tipo di programmi, quindi scriveremo prima una versione apparentemente corretta, ma che in realtà contiene diversi errori, e poi la miglioreremo un po' alla volta.

Il controllore della tastiera può essere programmato per generare una richiesta di interruzione quando dispone di un nuovo dato nel registro RBR. Per farlo occorre scrivere il byte 0x60 nel registro CMR (indirizzo 0x64 dello spazio di I/O) seguito dal byte 0x61 nel registro TBR (indirizzo 0x60 dello spazio di I/O). Una volta abilitate, le interruzioni possono essere disabilitate scrivendo 0x60 sia in CMR che in TBR. Dopo aver inviato una richiesta di interruzione, il controllore non ne invia una nuova fino a quando il software non legge il registro RBR.

In Figura 1 vediamo la prima versione del codice. L'idea è di fare in modo che la funzione `c_driver()` (righe 10–14) venga eseguita ogni volta che il controllore della tastiera invia una richiesta di interruzione. Per farlo è necessario sapere a quale piedino del controllore APIC la tastiera è collegata: nel nostro caso è il piedino numero 1. Quindi si deve scegliere una entrata libera della Interrupt Descriptor Table, per esempio la numero 40 (si noti che le prime 32 non sono utilizzabili, per motivi che vedremo) e scrivervi l'indirizzo del codice da eseguire. Il numero 40 sarà il *tipo* (o vettore) dell'interruzione. Si deve infine configurare l'APIC in modo che possa inviare tale tipo alla CPU ogni volta che inoltra una richiesta di interruzione proveniente dal piedino 1.

Per eseguire queste operazioni utilizzeremo delle funzioni già presenti nella libreria: alla riga 18 inseriamo il puntatore alla funzione nell'entrata numero 40 della IDT. Si noti che non inseriamo direttamente il puntatore a `c_driver()`, ma passiamo prima dalla funzione `a_driver`, che chiama `c_driver()` e poi invoca `iretq`. Questo perché le routine di interruzione devono terminare con `iretq`, mentre sappiamo che le routine C++ terminano con `ret`. Si faccia attenzione a non omettere la `q`: l'istruzione `iret` esiste, ed è la versione a 32 bit che, nel nostro caso, non può funzionare.

Alla riga 19 creiamo l'associazione tra il piedino 1 e il tipo 40. La funzione `apic_set_VECT()` scrive nell'opportuno registro dell'APIC.

```
1  #include <libce.h>
2  const ioaddr iSTR = 0x64;
3  const ioaddr iTBR = 0x60;
4  const ioaddr iRBR = 0x60;
5  const ioaddr iCMR = 0x64;
6
7  bool nuovo_car;
8  natb c;
9  extern "C" void a_driver();
10 extern "C" void c_driver()
11 {
12         inputb(iRBR, c);
13         nuovo_car = true;
14 }
15
16 int main()
17 {
18         gate_init(40, a_driver);
19         apic_set_VECT(1, 40);
20         apic_set_MIRQ(1, false);
21         outputb(0x60, iCMR);
22         outputb(0x61, iTBR);
23
24         for (;;) {
25                 nuovo_car = false;
26                 while (!nuovo_car)
27                         ;
28                 if (c == 0x01)
29                         break;
30                 for (int i = 0; i < 8; i++) {
31                         char_write(c & 0x80 ? '1' : '0');
32                         c <<= 1;
33                 }
34                 char_write('\n');
35         }
36 }
```

```
1  .extern        c_driver
2  .global        a_driver
3  a_driver:
4                 call        c_driver
5                 iretq
```

Figura 1: Versione 1.

L'APIC può disabilitare ogni piedino in modo indipendente, e quanto il programma viene caricato, la libreria `libce` provvede a disabilitare tutti i piedini. Alla riga 20 riabilitiamo il piedino 1.

Infine, alle righe 21 e 20 programmiamo il controllore della tastiera in modo che invii richieste di interruzione. Da questo momento in poi, mentre la CPU sta eseguendo il ciclo 24–35, la pressione di un tasto causerà l'esecuzione del codice in 12–13. Dobbiamo pensare di avere ora a disposizione due flussi di controllo: quello normale, del programma principale, e quello della routine di interruzione, che si inserisce in modo imprevedibile in quello principale. Proprio per via di questa imprevedibilità, non abbiamo altro modo di passare informazioni tra il codice principale e la routine di interruzione se non tramite variabili globali. Questo è lo scopo delle variabili `c` (linea 8) `nuovo_car` (linea 7). La prima contiene il nuovo codice di scansione letto (linea 12) e la seconda serve a segnalare che un nuovo codice è stato letto. Il programma principale aspetta che `nuovo_car` diventi vera (ciclo 26–27) dopo averla resa falsa (riga 25). Se diventa vera, vuol dire che `c_driver()` è andato in esecuzione (riga 13) e dunque `c` contiene un nuovo codice di scansione. Il programma principale procede dunque a mostrarlo sul video (righe 30–34). (Si noti che, in questo esempio, non stiamo sfruttando in alcun modo la possibilità offertaci dal meccanismo delle interruzioni, cioè di poter svolgere altre elaborazioni mentre siamo in attesa di un nuovo dato.)

Se si prova ad caricare ed eseguire questo programma sulla macchina virtuale, noteremo subito un problema: la prima pressione di un tasto mostrerà la stampa del corrispondente codice di scansione, ma premendo altri tasti non verrà mostrato più niente. Il problema è che non abbiamo inviato l'End Of Interrupt al controllore APIC. All'avvio, la libreria ha configurato il piedino 1 con riconoscimento delle interruzioni "sul livello". Questo vuol dire che il controllore APIC riconosce una richiesta di interruzione quando il segnale sul piedino passa da 0 e 1, quindi smette di guardarlo fino a quando non riceve l'EOI (e a quel punto riconosce una nuova richiesta se lo trova ancora a 1)[1]

Corregiamo dunque il programma come in Figura 2. Invochiamo la funzione `apic_send_EOI()` prima di terminare la routine di interruzione (linea 15). La routine è dichiarata nel file `apic.h` della libreria, che includiamo alla linea 2.

Questo programma sembra ora funzionare, ma in realtà contiene diversi gravi errori.

Il primo tipo di errori si manifesta solo per particolari condizioni. Si tratta di errori molto insidiosi, che possono restare nascosti per molto tempo. Per vederli, dobbiamo ragionare in questo modo: ora l'esecuzione di `c_driver()` può inserirsi tra due qualunque delle istruzioni che vanno da 24 a 34. Siamo sicuri che in ogni caso il programma si comporterà correttamente? No. Che succede se `c_driver()` viene eseguita subito prima della riga 25? Il suo assegnamento a `nuovo_car` (riga 13) verrà sovrascritto da quello alla riga 25. Di fatto, il programma principale non si accorgerà del nuovo codice. Un errore forse ancora

[1] Anche se il piedino fosse stato impostato con riconoscimento "sul fronte", il mancato invio dell'EOI avrebbe creato lo stesso problema, ma per un motivo diverso che vedremo in seguito.

```
1  #include <libce.h>
2  #include <apic.h>
3  const ioaddr iSTR = 0x64;
4  const ioaddr iTBR = 0x60;
5  const ioaddr iRBR = 0x60;
6  const ioaddr iCMR = 0x64;
7
8  bool nuovo_car;
9  natb c;
10 extern "C" void a_driver();
11 extern "C" void c_driver()
12 {
13         inputb(iRBR, c);
14         nuovo_car = true;
15         apic_send_EOI();
16 }
17
18 int main()
19 {
20         gate_init(40, a_driver);
21         apic_set_VECT(1, 40);
22         apic_set_MIRQ(1, false);
23         outputb(0x60, iCMR);
24         outputb(0x61, iTBR);
25
26         for (;;) {
27                 nuovo_car = false;
28                 while (!nuovo_car)
29                         ;
30                 if (c == 0x01)
31                         break;
32                 for (int i = 0; i < 8; i++) {
33                         char_write(c & 0x80 ? '1' : '0');
34                         c <<= 1;
35                 }
36                 char_write('\n');
37         }
38 }
```

Figura 2: Versione 2.

```
1  #include <libce.h>
2  #include <apic.h>
3  const ioaddr iSTR = 0x64;
4  const ioaddr iTBR = 0x60;
5  const ioaddr iRBR = 0x60;
6  const ioaddr iCMR = 0x64;
7
8  bool nuovo_car;
9  natb c;
10 extern "C" void a_driver();
11 extern "C" void c_driver()
12 {
13         inputb(iRBR, c);
14         nuovo_car = true;
15         apic_send_EOI();
16 }
17
18 int main()
19 {
20         gate_init(40, a_driver);
21         apic_set_VECT(1, 40);
22         apic_set_MIRQ(1, false);
23         outputb(0x60, iCMR);
24         outputb(0x61, iTBR);
25
26         for (;;) {
27                 nuovo_car = false;
28                 while (!nuovo_car)
29                         ;
30                 if (c == 0x01)
31                         break;
32                 for (int i = 0; i < 8; i++) {
33                         char_write(c & 0x80 ? '1' : '0');
34                         c <<= 1;
35                         for (int j = 0; j < 1000000; j++)
36                                 ;
37                 }
38                 char_write('\n');
39         }
40 }
```

Figura 3: Versione 3.

peggiore si verifica se l'interruzione arriva durante il ciclo 32–35: la variabile `c` viene sovrascritta con un nuovo codice di scansione, mentre il programma principale sta ancora stampando il precedente. Per riuscire ad osservare il verificarsi di questo errore, aggiungiamo le righe 35–36 (Figura 3). In questo modo rallentiamo l'esecuzione del ciclo, rendendo molto più probabile l'arrivo dell'interruzione al suo interno. Sarà facile ora osservare che, premendo anche sempre lo stesso tasto, il programma può mostrare configurazioni di bit diverse.

Questi due errori hanno in comune lo stesso problema: non possiamo accettare la richiesta di interruzione in qualunque momento. In questo semplice esempio, in realtà, di fatto la possiamo accettare solo mentre il programma sta eseguendo le righe 28–29. In Figura 4 vediamo una possibile soluzione. Alle righe 13–14 disabilitiamo le richieste di interruzione dalla tastiera, *prima* di leggere dal registro RBR. In questo modo, anche se la tastiera ha un nuovo codice di scansione da inviare (per esempio, il break code del tasto che abbiamo appena premuto e rilasciato), non invierà una nuova richiesta, dandoci il tempo di processare il precedente. Alle righe 28–29 riabilitiamo le interruzioni, *dopo* aver scritto `false` in `nuovo_car` e prima di entrare nel ciclo 30–31. Ora possiamo ricevere una nuova richiesta, e cosìvia.

Con queste modifiche il programma sembra di nuovo funzionare, ma contiene in realtà ancora degli errori. Questo secondo tipo di errori si manifesta se abilitiamo le ottimizzazioni nel compilatore (opzione `-O`n con $1 \leq n \leq 3$). Per esempio, modifichiamo lo script `compile`, che si trova nella sottodirectory `CE/bin` della directory *home*, aggiungendo `-O2` alle opzioni contenute nella variabile `COMPILER_OPTIONS`. Se ora ricompiliamo e rieseguiamo, vedremo che il nostro programma non stampa più niente. Il problema è che il compilatore C++ non sa dell'esistenza delle interruzioni, e assume che niente possa interferire con l'esecuzione di una funzione. Osservando il ciclo 28–29 noterà che (secondo lui) niente può modificare `nuovo_car`. Quindi il `while` verrà tradotto così:

```
if (!nuovo_car)
    while (true); // ciclo infinito!
```

Dobbiamo informare il compilatore del fatto che la variabile `nuovo_car` può cambiare il proprio valore anche se niente sembra modificarla nel ciclo 30–31. Questo si ottiene dichiarandola `volatile` come nella riga 8 di Figura 5. Per sicurezza, dichiariamo `volatile` anche `c` (linea 9), in quanto anche questa è modificata da `c_driver()` senza il compilatore lo possa sapere. Si noti che questo comporta che non possiamo passare direttamente `c` come secondo argomento di `inputb()`, in quanto il C++ non consente di passare un tipo `volatile` ad un riferimento non `volatile`. È possibile però assegnare un non `volatile` ad un `volatile`, quindi ricorriamo alle modifiche alle linee 13 e 16–17.

Un ultimo problema è dato dall'uso dei registri da parte di `c_driver()`. Il compilatore non salverà e ripristinerà il valore dei registri `%rsi`, `%rdi` etc., perché per lui `c_driver()` è una normalissima funzione, che segue le regole di aggancio di tutte le funzioni C++. Il problema è che ora questa funzione può inserirsi tra due istruzioni qualunque del programma principale (se le interruzioni sono abilitate). Che succede se si inserisce tra il caricamento di uno di

```
1  #include <libce.h>
2  #include <apic.h>
3  const ioaddr iSTR = 0x64;
4  const ioaddr iTBR = 0x60;
5  const ioaddr iRBR = 0x60;
6  const ioaddr iCMR = 0x64;
7
8  bool nuovo_car;
9  natb c;
10 extern "C" void a_driver();
11 extern "C" void c_driver()
12 {
13         outputb(0x60, iCMR);
14         outputb(0x60, iTBR);
15         inputb(iRBR, c);
16         nuovo_car = true;
17         apic_send_EOI();
18 }
19
20 int main()
21 {
22         gate_init(40, a_driver);
23         apic_set_VECT(1, 40);
24         apic_set_MIRQ(1, false);
25
26         for (;;) {
27                 nuovo_car = false;
28                 outputb(0x60, iCMR);
29                 outputb(0x61, iTBR);
30                 while (!nuovo_car)
31                         ;
32                 if (c == 0x01)
33                         break;
34                 for (int i = 0; i < 8; i++) {
35                         char_write(c & 0x80 ? '1' : '0');
36                         c <<= 1;
37                         for (int j = 0; j < 1000000; j++)
38                                 ;
39                 }
40                 char_write('\n');
41         }
42 }
```

Figura 4: Versione 4.

```
1  #include <libce.h>
2  #include <apic.h>
3  const ioaddr iSTR = 0x64;
4  const ioaddr iTBR = 0x60;
5  const ioaddr iRBR = 0x60;
6  const ioaddr iCMR = 0x64;
7
8  volatile bool nuovo_car;
9  volatile natb c;
10 extern "C" void a_driver();
11 extern "C" void c_driver()
12 {
13         natb cc;
14         outputb(0x60, iCMR);
15         outputb(0x60, iTBR);
16         inputb(iRBR, cc);
17         c = cc;
18         nuovo_car = true;
19         apic_send_EOI();
20
21 }
22
23 int main()
24 {
25    ... // come in Figura 4
26 }
```

Figura 5: Versione 5, parte C++.

```
1  #include "libce.s"
2  .extern        c_driver
3  .global        a_driver
4  a_driver:
5                 salva_registri
6                 call        c_driver
7                 carica_registri
8                 iretq
```

Figura 6: Versione 5, parte assembler.

questi registri e il suo successivo uso? Il programma principale si troverà ad usare il contenuto scorretto dei registri. Per rimediare a questo, salviamo e poi ripristiniamo tutti i registri intorno alla chiamata di `c_driver()` in `a_driver` (righe 5 e 7 in Figura 6). Per farlo utilizziamo due *macro* definite nella libreria, che includiamo alla riga 1.